5-1-2011

A New Artificial Intelligence for Auralux

Edward McNeill
Dartmouth College

A New Artificial Intelligence for Auralux

Dartmouth Computer Science Technical Report TR2011-692

Edward McNeill

May 2011

1 Introduction

This project focused on developing a more challenging artificial intelligence for the real-time strategy game Auralux. This game features a simple rule set yet offers a wide range of strategies. The AI currently released with the game is easily exploitable for experienced players, indicating a need for a more flexible and intelligent AI. Because the number of possible moves and number of decision points in a continuous-time game is vast for each player, a search algorithm such as minimax, as is often used for games like Chess, is impractical. I therefore set out to write a challenging rule-based AI.

I designed and implemented an AI framework named FlexibleAI that could be configured with various parameters controlling different aspects of the overall algorithm. In this way, the AI could be tuned to be more successful. I then created a testing framework called AuraSim that simplified Auralux into an easily-simulated turn-based format. I then tested the effectiveness of various configurations of the FlexibleAI by simulating many games in which the FlexibleAI played against a set of baseline AIs.

I began by testing a very broad set of configurations and used the results to craft tests for more narrowly defined and more precisely optimal configurations. Through this process I was able to converge on a set of parameters that defined an AI with a high victory rate. Surprisingly, this AI featured many of the most simple components that were tested, and most of its success came from a few central strengths, such as its ability to use concentrated attacks and its carefully-tuned aggression parameters.

Although there are many possible ways of improving the resultant AI, I consider the project a success since it produced a more challenging AI that will likely prove more satisfying to play against. The current plan is to implement the FlexibleAI in Auralux, converting it into a real-time AI. With some adjustments to add variety and to account for unique real-time considerations, I expect to include this new AI in a sequel or update to Auralux.

2 Auralux

Auralux is a simple real-time strategy game for three players: one human player and two AI players. The game takes place on an infinite plane that is populated with two types of objects: suns and soldiers. Suns (named for their graphical representation in the game) are stationary objects of varying size that constantly produce soldiers every second for the player that owns them. Soldiers, in contrast, are under the direct control of the player and can be moved anywhere. Soldiers are used by the player to destroy enemy soldiers, capture enemy suns, upgrade suns to increase their rate of production, and eventually control the entire game area. This minimal rule set provokes a wide range of strategies, which vary drastically in effectiveness.

I developed Auralux and released it in January 2011. Since then, the game has been downloaded over 65,000 times and received positive reviews. One common complaint, however, was the the AI was too simple. While it would offer a satisfying challenge to new players, it was too easy to see through and exploit the AI's actions. For example, the AI would not distinguish between its opponents, and so the player could lead two two AIs to fight each other and leave the player unscathed. Further, the AI only considered actions from the perspective of a single sun at a time, leaving it with no understanding of the state of the game as a whole. Experienced players could essentially solve the AI and beat it consistently.

Since then I have continued work on bringing Auralux to new platforms and preparing for the development of a sequel. As part of this effort, I began development of a new and more challenging AI. To begin this process, I developed a testing framework to help determine which algorithms and parameters formed the most effective AI.

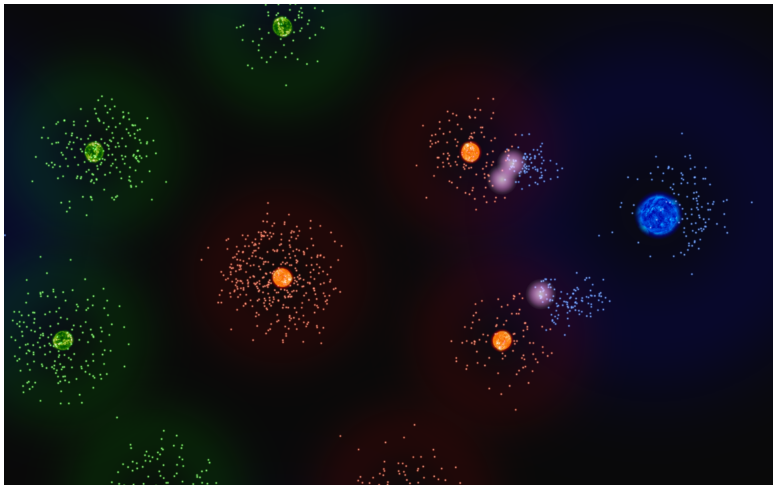

Figure 1: A screenshot from the released version of Auralux.

3 AuraSim

In order to test new AI strategies, I developed a simulation of Auralux called AuraSim. AuraSim featured several simplifications that allowed for much faster game simulation while preserving the rules and dynamics of Auralux.

The most important of these simplifications was the discretization of the game's playing area into an undirected graph. The suns form nodes on the graph, and soldiers exist within suns. This is an accurate model of how most games of Auralux are played, with soldiers only traveling between nearby suns. Each graph (referred to as a level) is rotationally symmetrical in order to ensure fairness between the players. AuraSim features four levels, each of which correlates to an equivalent existing level in Auralux. As is the case in Auralux, each sun in the level is given a maximum size, from one to three, and a sun cannot be upgraded past this size.

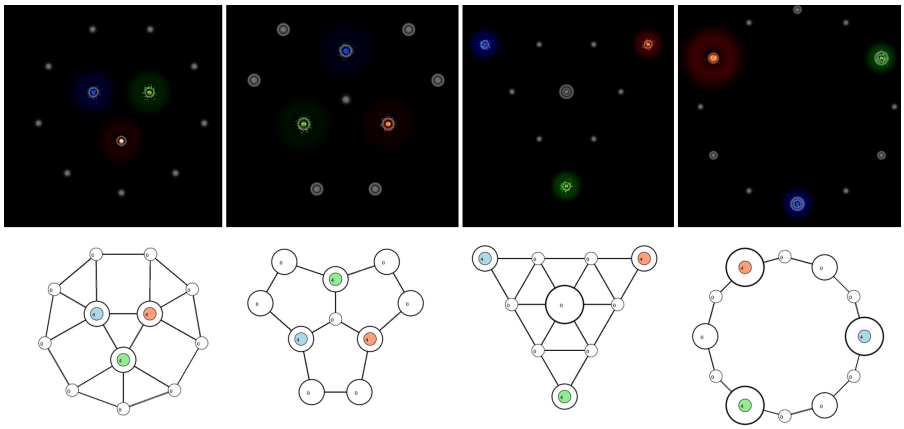

Figure 2: Examples of Auralux levels alongside their AuraSim equivalents

A second simplification was the discretization of time into simultaneous turns. Each turn represents the time it would take for a soldier to travel from one sun to an adjacent sun. The turns are divided into two phases: first each player chooses what to do with their soldiers, and secondly these actions are resolved.

The last major simplification was the condensing of soldiers into groups such that one soldier in AuraSim equates to twenty-five soldiers in Auralux. This is approximately the number of soldiers that a size-one sun would produce in a single turn.

On the first phase of each turn, a player may choose for each soldier whether to move that soldier to a linked sun, upgrade the soldier's current sun, or stay idle. These simple actions are resolved based on their context. When a sun has had four soldiers upgrade it (which destroys the soldiers), it increases in size,

thereby increasing its rate of soldier production. If two opposing soldiers meet each other when moving between suns, they are both destroyed. If more than one player has soldiers on a sun after movement, the soldiers destroy each other one-for-one. If the remaining soldiers belong to an enemy of their new sun, they will attack the sun. Once four soldiers have attacked a sun, it is destroyed, becoming a size-zero empty sun. If soldiers remain on an empty sun, they will automatically try to colonize it, i.e. upgrade it. After both phases of the turn are complete, each sun produces a number of soldiers equal to its size. Once a player has captured all suns, that player wins.

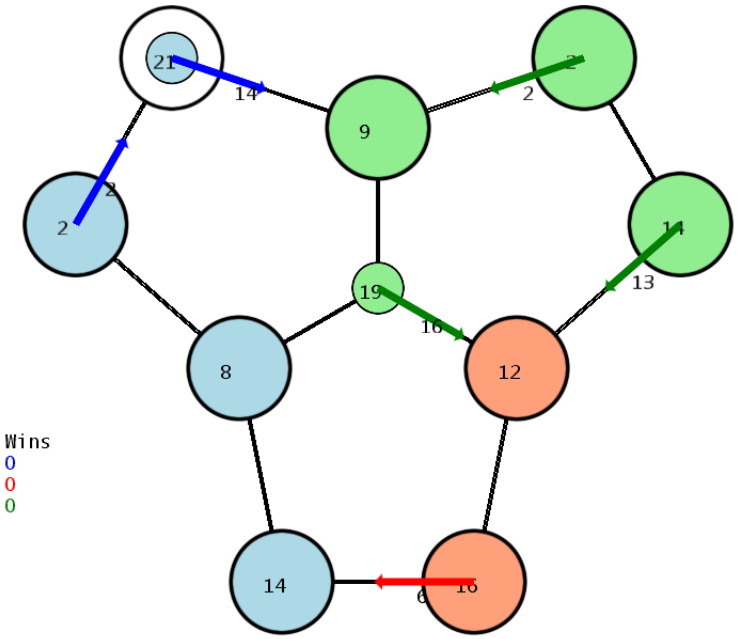

Figure 3: A screenshot of AuraSim showing the players' current moves.

To recap the core rules:

- Each turn, soldiers may upgrade their current sun or move along a link to another sun.
- When two opposing soldiers meet, they destroy each other.
- Spending four soldiers destroys an enemy sun (bringing it to size zero).
- Spending four soldiers upgrades a friendly sun (increasing its growth rate by one).

There are a few complications that may impair the accuracy of the simulation in its current form. The most obvious of these is the presence of race conditions in AuraSim. When all three players try to colonize a sun with equal force, for example, the first two players to arrive will cancel each other out, while the third will take it unchallenged. This strategic situation occurs in Auralux as well. The simulation currently handles these conditions by randomizing which players' soldiers resolve their attacks first. While this does not perfectly model Auralux (in which the results would be dependent on who attacked first and decided based on a difference of a few seconds), it offers a satisfying compromise that does not privilege any player.

In other situations, the issue is not a race condition, but rather real-time changes in strategy. Consider the situation in which two players are simultaneously attempting to colonize an empty sun. Each player is hoping that the other will cancel its attack, since destroying each others' soldiers to no effect would be the worst outcome for both, but if the other cancels the attack, then the player would be best served by continuing. This situation is a classic game of chicken, and while the equilibrium strategy (a 50/50 randomized mixed strategy) can be easily modeled, this does not represent the true range of possibilities in the real-time game. AuraSim does not allow for these last-moment changes to orders, and so all attacks are carried out to completion. While uninterrupted attacks are not optimal play in Auralux, they tend to offer a more satisfying real-time game experience for a human player (who likely doesn't enjoy playing endless games of chicken with a computer) and so satisfy the goals of this project, though this disconnect between optimal play and enjoyable play can be considered a flaw in the game design of Auralux.

Another complication involves attacks that do not occur on the paths directly between nearby suns. These roundabout attacks are allowed in Auralux and caused one of the major weaknesses of the released game's AI, and they are not modeled in the current AuraSim graph structure. However, such attacks are rarely useful from a purely strategic standpoint if all players are aware enough to defend against them, and such defenses can be easily incorporated into a real-time re-implementation of the AIs produced in this project. Therefore they can be safely ignored within the context of the simulation.

4 Baseline AI Strategies

An AI strategy is a set of rules that determines a player's actions during the first phase of each turn. For this project, I created one highly configurable AI strategy (FlexibleAI) and implemented three other AI strategies (RandomAI, AggressiveAI, and ReleasedAI) to provide a consistent basis for comparison. Each of these three AIs operates by examining each sun and determining from that perspective what to do with that sun's soldiers, a limited perspective that hinders their overall effectiveness.

4.1 RandomAI

The RandomAI strategy chooses randomly whether to use its soldiers to upgrade its suns, move towards the enemy (sometimes resulting in an attack), or stay idle. It provides a simple baseline AI.

```
for each sun S owned by this player:
  if S can upgrade:
    50% chance:
      S upgrades
  50% chance:
    S moves all soldiers to a random sun that is closer to the enemy
```

4.2 AggressiveAI

The AggressiveAI strategy always attacks if it is next to an enemy. Failing that, it prefers to expand outwards, colonizing empty suns, rather than upgrade existing suns. Only when there are no empty suns or enemy suns adjacent will a sun upgrade. This simplistic strategy would be well-suited to a two-player zero-sum game, but tends to do poorly at three-player games since it divides its forces between its two opponents.

```
for each sun S owned by this player:
  if S is adjacent to an enemy sun:
    S attacks the enemy sun with all available soldiers
    return
  else: //not adjacent to an enemy sun
    if S is adjacent to an empty sun:
      S colonizes the empty sun
      return
    if S is not fully upgraded:
      S upgrades
    else:
      S sends all available soldiers to a friendly sun
        that is closer to an enemy sun
```

4.3 ReleasedAI

This AI strategy is adapted from the AI that is currently associated with Auralux in its originally released form. It was designed to offer a satisfying and flexible challenge, but was ultimately limited by its sun-by-sun perspective.

```
for each sun S owned by this player:
  if S is adjacent to an enemy sun:
    if S has many soldiers relative to its enemy neighbors,
      and S has enough soldiers to destroy its weakest neighbor:
      S attacks its weakest neighbor with all available soldiers
    if S has enough soldiers to upgrade or colonize a neighboring empty sun
      40% chance:
        S colonizes a random adjacent empty sun
      otherwise:
        S upgrades
    if S is fully upgraded and has many soldiers to spare:
      S attacks a random enemy sun with few soldiers
  else: //not adjacent to an enemy sun
    if S has any adjacent empty suns:
      25% chance:
        S upgrades if possible
      otherwise:
        S colonizes the neighboring empty sun
    if S has enough soldiers to upgrade:
      S upgrades
    if S is fully upgraded and has no adjacent empty suns:
      S sends all available soldiers to an adjacent friendly sun
        that is closer to an enemy sun
```

This simple strategy has several compelling advantages from a game design perspective. First, it provides just enough randomness to provide variety between play sessions. This randomness also offers the illusion of personality or improvisation in the AI. Second, the ReleasedAI is aggressive and mobile, constantly moving and growing towards the player. Since the game begins with such symmetry and equality between players, this aggression is enough to provide a fun challenge that can still be overcome by clever strategy. However, I have observed that once players discover the algorithm's general weaknesses, they have essentially "solved" the game's puzzle, rendering any future challenge impossible. These weaknesses must be addressed in order to keep the game's strategic challenge fresh.

The first major weakness of the ReleasedAI is that it has no understanding of which player is winning at any moment; since it can only examine the danger of the sun it is currently evaluating, no two players could ally to defeat a third, winning player. A second major weakness is that the AI will not allocate its resources intelligently. The AI will move its soldiers towards its enemies, but it will not try to build up forces for useful attacks nor reinforce imperiled suns. A

third weakness is that, due to the sun-by-sun perspective of the strategy, the ReleasedAI will never use the soldiers from multiple suns in a single, coordinated attack. The focus of this project, the FlexibleAI strategy, is meant to address these weaknesses.

5 FlexibleAI

In order to improve on the ReleasedAI, it was necessary to create an algorithm that would make decisions based on the entire game state, rather than only considering the situation immediately surrounding a sun. The FlexibleAI strategy is an attempt at such a solution. The strategy operates by proceeding through these steps:

1. Calculate a Value for each sun. This represents the strategic value of the sun to this faction.
2. Calculate a Danger for each of this player's suns. This represents the number of soldiers projected to be necessary to successfully defend the sun.
3. Move soldiers towards defensive posture that best accommodates the danger on each sun.
4. Designate an enemy sun as a target.
5. Determine whether or not to upgrade each sun.
6. Move soldiers that are not defending towards the target unless the soldiers are already adjacent.
7. Evaluate a possible attack upon the target. If likely to succeed, execute the attack.
8. Evaluate extra attacks against enemy suns other than the target. If likely to succeed, execute the attack.

This provides a simple means of determining a global defensive posture, allocating soldiers where they will provide the most advantage and concentrating force during an attack.

There are some disadvantages to this strategic framework. First, it only considers one target at a time. While this is usually advantageous due to the need to concentrate soldiers during attacks, in some situations it is less so. For example, when the faction's suns are split from each other, perhaps on opposite ends of the level, forces cannot always gather to attack the single most valuable target. In other cases, it might be more advantageous to split up forces in anticipation of striking in several places at once, such as the situation in which an enemy faction is compelled to draw away forces from many suns at once.

Second, the framework always privileges defense. Soldiers are only free to attack or move if they are not first assigned to defend. While this ensures that the AI does not open any unnecessary vulnerabilities, it can fail to exploit some highly valuable attacks that would require a sacrifice of a less valuable sun. Defense is also a sub-optimal strategy when the sun is vastly outnumbered by adjacent enemies. In this situation, retreat or consolidation of soldiers into fewer suns is more likely to result in later advantages.

Lastly, the framework remains simple and rule-based, with no considerations for the flow of soldiers through a level, enemy reactions to attacks, possible future game states, or idiosyncrasies of different map structures. It is ultimately a simple and greedy strategy.

However, the strategy proved to be effective despite these disadvantages. I implemented a simple and straightforward version of the FlexibleAI in order to examine its potential. This version, despite obvious opportunities for improvement, achieved nearly double the victory rate of the ReleasedAI algorithm. The broad advantages of globally aware soldier allocation and concentrated attacks were enough to demonstrate the relative power of this framework.

The FlexibleAI also provides an excellent platform for experimentation and revision. The various steps of the strategy can be altered and tested quickly, eventually moving towards a single very powerful algorithm. In order to explore and optimize the FlexibleAI strategy, the strategy was broken down into its component sections. These strategy components can be altered or replaced independent of each other. I attempted to approach the optimal FlexibleAI settings by testing different combinations of strategy components. The strategy was separated into the following components:

5.1 Valuation Parameters

This component comprises six floating point numbers, represented as doubles, that determine the value of each sun and thereby determine which suns the strategy prefers to attack first. They are:

- sunBuiltSizeValue: a floating point number representing the value of a sun of size one. This scales with the size of the measured sun. For example, if this parameter was set to four, then a sun of size two would be valued at eight. If this parameter is large, then the strategy will focus on attacking larger suns. If it is small, then the strategy will mostly disregard the size of suns.

- sunPotentialSizeValue: a floating point number representing the value of the unrealized potential sun size. E.g. a sun of size one and a max size of three would have this parameter times two added to its value. If this parameter is much larger than the Sun Build Size Value, then the strategy will care more about pursuing future growth potential rather than attacking large enemy suns.

- friendlyConnectionValue: a floating point number representing the value added to a sun for each neighbor sun that belongs to this strategy's faction. If this parameter is positive, then the strategy will prefer to attack suns that are safely surrounded by friendly suns. If negative, then the strategy will avoid such attacks.

- enemyConnectionValue: a floating point number representing the value added to a sun for each neighbor sun that belongs to an enemy of this strategy's faction. If this parameter is positive, then the strategy will prefer bold attacks against suns that are surrounded by enemies. If negative, then the strategy will avoid such attacks.

- advantagedPreferenceValue: a floating point number that scales the value of enemy suns when the strategy's faction is winning the game by a significant margin. The second-place enemy's suns are scaled directly by this parameter, while the third-place enemy's suns are scaled by this parameter's multiplicative inverse. If this parameter is greater than one, then the strategy will prefer to attack its stronger rival when winning the game. If the parameter is smaller than one, then the strategy will prefer to first extinguish its weaker rival.

- disadvantagedPreferenceValue: a floating point number that scales the value of enemy suns when the strategy's faction is losing the game by a significant margin. The winning enemy's suns are scaled directly by this parameter, while the other enemy's suns are scaled by this parameter's multiplicative inverse. If this parameter is greater than one, then the strategy will prefer to attack the winning enemy if losing. If the parameter is smaller than one, then the strategy will prefer to attack the other losing faction first.

5.2 Danger Projection

This component comprises a floating point number (dangerModifier) and a choice of three different methods for projecting danger from enemies onto this strategy's suns: to all suns equally equally, to the most connected suns, or proportionally by value. This choice will affect how the strategy prefers to allocate soldiers in defense.

- The dangerModifier is a floating point number that scales the danger of each of this strategy's suns. This determines the level of protection that it takes for the strategy to consider a friendly sun to be defended. If this parameter is large, the strategy will tend to be more conservative, while if this parameter is small, the strategy will be more reckless.

- Projecting danger equally involves determining the number of this strategy's suns that are linked to the enemy sun and splitting that sun's current strength equally among all the friendly linked suns. This is a simple projection that reflects the wide range of possible enemy attacks.

- Projecting danger to the most connected suns involves projecting each enemy sun's strength onto the friendly sun that neighbors the most other friendly suns. This projection emphasizes a centralized defense, in with the strategy protects the suns that offer the most future flexibility.

- Projecting danger proportionally by value involves projecting the strength of each enemy sun upon each neighboring friendly sun in an amount proportional to the friendly suns' relative value. This emphasizes the defense of the most valuable suns.

5.3 Defense Allocation Method

This component is defined by a choice of three different methods for moving soldiers into a defensive posture: defend lazily, by value, or by most danger. This choice determines how the strategy moves its units into the preferred defensive arrangement.

- Defending lazily means that soldiers will not leave a sun if that would leave the sun with fewer soldiers than its assigned danger. Excess soldiers will move to defend the highest-value undefended neighboring suns. This ensures that defenses are put to effective use as quickly as possible.

- Defending by value means that soldiers will prioritize defenses of itself and its neighbors in order of descending value. Only after the higher-value suns have been defended will the lower-value suns receive reinforcements. This emphasizes the defense of the most valuable suns.

- Defending by danger means that soldiers will prioritize defenses of itself and its neighbors in order of descending danger. This offers a more direct means of countering large attacks.

5.4 Attack Targeting Method

This component is defined by a choice of three methods for choosing an enemy sun to be the primary target of the strategy: choosing the target by value, by advantage, or by vulnerability. This choice will determine whether the strategy is generally focused on high-value targets or is opportunistic in its targeting.

- Picking a target by value selects the most valuable sun that borders any of this strategy's suns. This is a straightforward focus on valuable property.

- Picking a target by advantage selects the neighboring enemy sun that has the most value subtracted by the number of soldiers defending it. This is an attempt to find the most efficient use of soldiers.

- Picking a target by vulnerability selects the neighboring enemy sun that has the fewest soldiers defending it. This disregards the calculated values and chooses targets opportunistically.

5.5 Upgrade Safety

- This component is defined by a floating point number (upgradeSafety), represented as a double, that determines what portion of the sun's danger must be defended against before the sun will upgrade. When this parameter is close to one, a sun will only upgrade when fully defended. When close to zero, suns will upgrade regardless of danger.

5.6 Positioning Method

This component determines whether nondefending soldiers move towards the target in preparation for attack or instead move towards the nearest enemy.

- First, the component is defined by an integer (targetPreference) that determines whether nondefending soldiers will move towards the target sun or towards the nearest enemy. Nondefending soldiers will move towards a target sun by default, but will instead move towards the nearest enemy soldier if that path is this number of suns shorter. A large positive parameter means that soldiers will always move towards the target, while a large positive parameter means that the soldiers will always move towards the nearest enemy.

- In addition, this component is defined by a boolean (stickToEnemy) that determines whether or not nondefending soldiers that are adjacent to an enemy can move. If true, nondefending soldiers next to an enemy sun may not move from their sun, even in order to move towards the target. If false, nondefending soldiers may move normally.

5.7 Attack Requirements

This component is defined by a choice of two methods for determining whether or not to launch an attack on the target sun: a simple attack, or a conditional attack. A conditional attack is defined by two floating point numbers, attackCushion and defenseCushion.

- The simple attack uses all available (i.e. nondefending) soldiers to attack the target whenever possible. This manifests as constant aggression.

- The conditional attack depends on two floating point numbers as parameters: defenseCushion and attackCushion. The algorithm allows defending soldiers to abandon their defense and participate in the attack. The attack will commence only if three conditions are met. First, the attack must be projected to succeed, with enough soldiers left over to colonize the target sun. Second, the attack must leave enough soldiers on the conquered sun to cover the sun's danger * attackCushion. Third, the attack must leave each attacking sun with at least their danger * defenseCushion soldiers left. A large attackCushion means that the strategy will attack only when

it can hold the target, while a small attackCushion means that the strategy will attack as soon as it has a good chance of destroying the enemy. A large defenseCushion means that no attack will go through unless defenses can be maintained. A small defenseCushion means that the attacking suns will abandon their defenses to go through with an attack.

5.8 Extra Attack Requirements

- Another choice of methods must then be made for extra attacks, i.e. attacks against enemy suns other than the main target. Each of the strategy's suns considers each of its neighbors for an attack, using either a reckless attack, a conditional attack, or an individual attack method. The reckless and conditional attacks are identical to those for the main target attack (with parameters extraAttackCushion and extraDefenseCushion). The individual attack permits only a single sun to commit soldiers to the attack, which allows more attacks per turn but disallows concentration of force during an attack.

6 Tests and Iteration

In order to determine the most powerful configuration of FlexibleAI parameters, I used AuraSim to simulate millions of games featuring thousands of different configurations. The basic formulation of the tests began with a selection of enemy AI strategies and a selection of levels to be tested. AuraSim would then run simulations against each enemy AI strategy on each level and add up the total number of victories for the tested AI configuration. This victory count functioned as that configuration's score relative to the others in that test.

On some occasions, simulated games reached a stalemate. It was very possible for two similar AIs to reach a point in which neither could gain any advantage over the other. Therefore, if a game went beyond 500 turns (well beyond the time it takes for most to conclude decisively), the game was declared a stalemate and the victory went to no player. Because stalemates are not desirable from a game design perspective, these were not replaced with an extra game in the test, and so dragged down the score of the AI strategy being tested.

Because the various parameters in the FlexibleAI strategy can depend upon each other in affecting the performance of the AIs (for example, the defenseCushion parameter is dependent on the earlier danger projection), a local search was not a practical means of determining an optimal AI. However, a comprehensive search of the different combinations of parameters was also out of the question due to the vast number of possibilities. Therefore, the testing began with an initially wide battery of tests, which was used to exclude many of the obviously suboptimal configurations. After that, I made more narrow tests and tweaks to the more promising results.

Each FlexibleAI configuration was tested on four levels and on four baseline AIs: the RandomAI, AggressiveAI, ReleasedAI, and a configuration of the Flex-

ibleAI called DefaultAI, which contained a set of moderate parameters. In each of these sixteen situations, the tested FlexibleAI would go through a certain number of simulated games. The AI's score was defined as its total number of victories in all of these games. Each of the tests was sequential, with the results of the previous test determining which FlexibleAI configuration is tested next.

For initial reference, these baseline AIs were each tested against each other. The player AI played versus two identical opponent AIs, and each AI played 10,000 games against each other, split across all levels.

	vs. Random	vs. Aggro	vs. Released	vs. Default
RandomAI	33.45%	37.23%	5.17%	3.74%
AggressiveAI	28.72%	24.8%	6.60%	6.34%
ReleasedAI	70.64%	72.99%	33.57%	20.51%
DefaultAI	79.99%	77.22%	58.60%	22.39%

Figure 4: Victory rates of baseline AIs

6.1 Broad Cautiousness Testing

The first test was designed to determine the best range of settings for the most important and wide-reaching parameters, primarily those that determine how cautious or aggressive the AI is. Each AI was tested in 1600 games evenly distributed between the various levels and baseline AIs. The tested parameters were the dangerModifier, targeting method, upgradeSafety, attack requirements, and extra attack requirements. The remaining parameters were equal to those in the DefaultAI.

Score	danger	target	upgrade	attack	defense	exAtk	exDef
1116	0.5	value	0.0	1.0	0.5	1.0	0.5
1110	0.5	value	0.0	1.0	0.25	1.0	0.5
1110	0.5	value	0.0	0.75	0.75	1.0	0.5
1110	0.5	value	0.5	0.75	0.75	1.0	0.5
1093	0.5	advntg	0.0	1.0	0.5	1.0	0.5
1087	0.5	value	0.5	0.5	0.75	1.0	0.5
1084	0.5	value	0.5	1.0	0.75	1.0	0.5
1080	0.5	value	0.0	1.0	0.75	1.0	0.5
1078	0.5	value	0.5	0.75	0.5	1.0	0.5
1077	0.5	value	0.5	0.0	0.75	1.0	0.5

Figure 5: Top ten AI configurations after Broad Cautiousness Testing

The top scoring AIs displayed considerable uniformity despite the noise in the data. Among the top ten AIs tested, all had a mid-level dangerModifier, all used

a conditional attack and extra attack method, all had an extraAttackCushion of 1, all had an extraDefenseCushion of 0.5, and all but one chose targets by value. The upgradeSafety tended towards the lower range, the attackCushion tended toward the higher range, and the defenseCushion tended towards the mid-to-high range. These trends were in evidence for most successful AIs.

6.2 Narrow Cautiousness Testing

Based on the results of the first test, these parameters were tested again with more finely-graded settings that clustered around the previous top results. To simplify the tests and allow for more samples, the extra attack requirement parameters were set as equal to the main attack requirement parameters. Further, all configurations used a targeting-by-value method and conditional attacks. The number of games player per AI was increased to 4,000 in order to reduce the impact of noise from the random elements in the algorithms.

Score	danger	upgrade	attack	defense
2730	0.4	0.5	1.1	0.6
2717	0.35	0.2	1.1	0.7
2717	0.4	0.2	1	0.6
2715	0.35	0.4	1.1	0.7
2712	0.35	0.5	1	0.8
2707	0.4	0.6	0.9	0.6
2707	0.4	0.6	1	0.6
2705	0.45	0.5	1.1	0.6
2703	0.45	0.2	1	0.6
2702	0.35	0.6	1	0.8

Figure 6: Top ten AI configurations after Narrow Cautiousness Testing

The top AI at this point was able to achieve a 68.25% victory rate, over four times that of its average opponent. Based on these results, values for each parameter were selected from the middle of each cluster of successful values. The resultant AI featured the following parameters:

```
dangerModifier = 0.4
attackCushion = 1.1;
defenseCushion = 0.7;
extraAttackCushion = 1.1;
extraDefenseCushion = 0.7;
upgradeSafety = 0.5;
```

This AI is one that generally tends to be aggressive, with a fairly low dangerModifier. However, it does not attack unless it can leave some defending soldiers with its attacking suns, and it will only attack with a large force that has a good chance of holding the target from enemy counterattacks.

6.3 Upgrade Safety Testing

Due to the significant variability in the upgradeSafety parameters in the top results of the previous tests, I chose to focus on this parameter for the third test. Due to the presumably slight differences between the parameter's effects, each AI was tested in 160,000 games each.

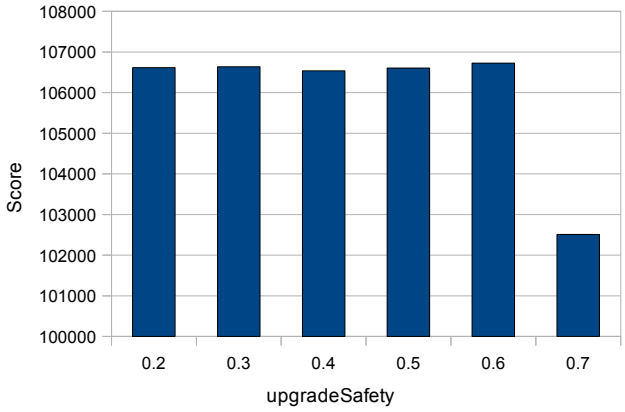

Figure 7: The scores of the FlexibleAI using upgradeSafety settings.

Each upgradeSafety level from 0.2 to 0.6 produced nearly identical results, and so it was left at 0.5 for future tests. This parameter, within these bounds, appears to have little influence on the results of the AI. While this is somewhat counterintuitive, note that soldiers that are not used for upgrading can be used to colonize empty suns, thereby increasing the player's production rate through attacks. These soldiers may also allow the player to attack more quickly, balancing out the advantage that would be gained from a quicker upgrade.

6.4 Positioning Method Testing

The fourth test took the earlier parameters and tested different soldier-positioning preferences with them. The targetPreference parameter (a soldier's threshold for preferring moving towards the target or towards the nearest enemy) was tested at values -100, -2, -1, 0, 1, 2, and 100, and each value was tested with the stickToEnemy parameter set to true and false. Each AI was tested in 16,000 games.

Surprisingly, the AI performs significantly better when ignoring the target during positioning. The results suggest that it is more useful to have soldiers reach the enemy as quickly as possible than to carefully position the soldiers in an optimal attack position. It is also apparent that the stickToEnemy parameter ceases to affect the AI's behavior once the AI prefers to move soldiers towards the nearest enemy. With the positioning method set to move towards the enemy

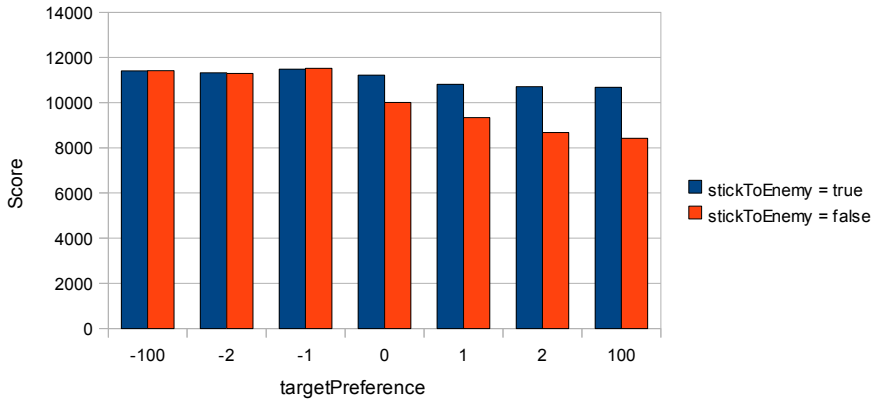

Figure 8: The scores of the FlexibleAI using different positioning methods.

as quickly as possible, the AI achieves a victory rate of 72%.

6.5 Defense Allocation Method Testing

The fifth test compared the three different methods of allocating units to match the danger associated with each sun. These methods were: defending lazily, by value, or by danger. Each AI was tested in 40,000 games.

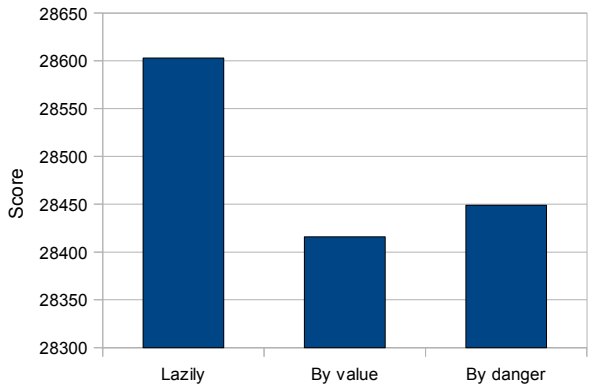

Figure 9: The scores of the FlexibleAI using different defense allocation methods.

As with the upgradeSafety parameter, this defense allocation method seems to have little effect on the success of the AI. This is probably because all methods

move towards the same generally-achievable goal of total defense cover. It is possible that since the AI has a low dangerModifier to begin with, the AI rarely has to choose which suns must be left undefended, and so the defense allocation method rarely comes into play. The lazy allocation method was selected for future use due to its simplicity and very slight advantage.

6.6 Main Attack Preference Testing

Because the settings for main attacks and extra attacks were equal in the previous tests, the next test focused on considering slight differences between them. When the two are equal, there is very little preference given towards the carefully-targeted main attack; it is evaluated first, but has no other advantage. This test compared various settings of the attackCushion and defenseCushion parameters as well as slight increases of the extraAttackCushion and extraDefenseCushion relative to them. That is, the test evaluated giving the main attack a relatively lower threshold before attack. Each AI was tested in 4,000 games.

Score	attack	defense	exAtk	exDef
2876	1.0	0.5	1.2	0.7
2873	0.9	0.5	1.0	0.6
2859	1.0	0.6	1.0	0.6
2857	1.0	0.7	1.2	0.9
2855	0.9	0.7	0.9	0.7
2854	1.0	0.6	1.2	0.8
2840	0.9	0.5	0.9	0.5
2838	1.0	0.6	1.1	0.7
2837	0.9	0.6	1.1	0.8
2831	0.9	0.6	1.0	0.7

Figure 10: Top ten AI configurations after Main Attack Preference Testing

While the results of this test are hardly dramatic, they do show some tendency towards slightly more aggressive main attacks and slightly more cautious extra attacks. Still, it is clear that these values do not have a major effect on the AI's scores. The following values were selected for the AI:

```
attackCushion = 1.0;
defenseCushion = 0.6;
extraAttackCushion = 1.2;
extraDefenseCushion = 0.7;
```

6.7 Danger Projection Method Testing

The seventh test compared the various methods of projecting danger: equally, to the most connected suns, or proportionally by value. Each AI was tested in 16,000 games.

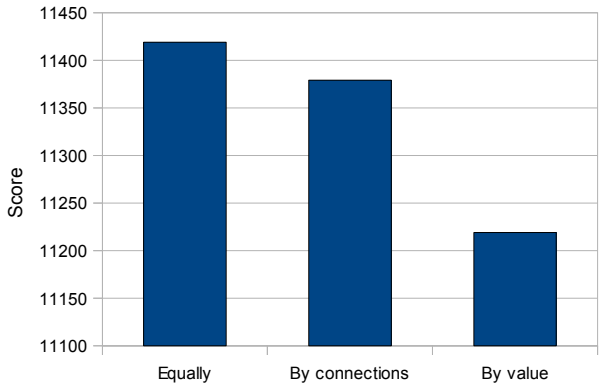

Figure 11: The scores of the FlexibleAI using different danger projection methods.

Here too a component was found to have little effect and to ultimately work best in its simplest form. It is possible that the only truly important factor is the number of soldiers kept at the front lines, and the distribution of them has little impact, since even poorly distributed units can be used for counterattacks.

6.8 Broad Valuation Parameter Testing

This test focused on the six valuation parameters (sunBuiltSizeValue, sunPotentialSizeValue, friendlyConnectionValue, enemyConnectionValue, advantagedPreferenceValue, and disadvantagedPreferenceValue) that determine how the AI judges the relative values of suns. They were considered together because they all affect a single value for each sun and therefore are naturally interdependent. Each AI was tested in 1,600 games.

Score	built	potential	fCon	eCon	advntg	disadvntg
1191	5	9	3	-3	1	1
1187	5	5	-3	-3	2	0
1183	1	5	9	-3	1	2
1180	5	9	9	-3	2	1
1177	1	5	3	-3	2	2
1176	5	9	9	-3	0	2
1175	5	5	3	-3	1	1
1174	1	5	3	-3	0	2
1174	5	5	3	-3	2	0
1173	1	5	3	-3	1	1

Figure 12: Top ten AI configurations after Broad Valuation Parameter Testing

The results of this test featured substantial noise, but nevertheless there was some uniformity in the top-scoring AIs. Further, there was a substantial difference between the best victory rate (74.4%) and worst (63.4%) in the set, indicating that these parameters have a significant effect. Among the top configurations, the enemyLinkValue was consistently negative, indicating that the AI performed best when attacking suns that were not connected to many enemy suns. Further, the sunPotentialSizeValue tended to be higher, indicating that a focus on suns with large growth potential was advantageous.

6.9 Narrow Valuation Parameter Testing

Because the previous test was heavily influenced by noise, a second valuation parameter test was conducted. This test excluded the advantage preference values and tested a new range of values, informed by the results of the broad test. Each AI was tested in 16,000 games.

Score	built	potential	friendlyCons	enemyCons
11735	7	7	3	-6
11735	7	11	6	-6
11715	4	7	6	-6
11708	4	7	3	-6
11696	4	7	9	-6
11673	1	11	6	-6
11670	4	11	9	-6
11669	4	7	6	-3
11666	7	11	9	-6
11650	7	11	3	-6

Figure 13: Top ten AI configurations after Narrow Valuation Parameter Testing

Again, a sizable gap was found between the best and worst AI victory rates (73.3% and 67.8%), and several weak trends were in evidence among the top scores. The top scores tended to exhibit a middling sunBuiltSizeValue, a large sunPotentialSizeValue, and a large negative enemyConnectionValue, while the friendlyConnectionValue was highly variable. Based on these results, the following values were selected:

```
sunBuiltSizeValue = 5;
sunPotentialSizeValue = 9;
friendlyConnectionValue = 6;
enemyConnectionValue = -6;
```

6.10 Advantage/Disadvantage Attack Preference Testing

The previous test excluded the advantage preference values, which control which faction the AI prefers to attack when it is winning (advantagedPreferenceValue) and losing (disadvantagedPreferenceValue). Therefore the final testing attempted to determine whether these values had any effect, and what their optimal values may be. Each AI was tested in 40,000 games.

Score	advantagedPreference	disadvantagedPreference
29130	1	3
29078	1	1
29074	1	2
29060	2	3
29055	2	0
29041	2	1
29039	2	2
29008	3	1
29001	3	2
28960	3	3

Figure 14: Top ten AI configurations after Advantage/Disadvantage Attack Preference Testing

The top AIs tend towards a low advantagedPreferenceValue while the disadvantagedPreferenceValue is more variable. It is also important to note that the victory rate of the best and worst AIs differ by less than one percentage point, indicting that these values are not very influential. The selected values for the final AI were:

```
advantagedPreferenceValue = 1.0
disadvantagedPreferenceValue = 3.0
```

7 Evaluation

The result of the entire battery of tests and revisions is a FlexibleAI configuration that achieves a victory rate of nearly six times its average opponent. This is unquestionably a major improvement over the ReleasedAI and should provide a more satisfying challenge for players. It is certainly possible to achieve a more precisely optimal configuration, but this is a successful outcome regardless.

	vs. Random	vs. Aggro	vs. Released	vs. Default
RandomAI	33.45%	37.23%	5.17%	3.74%
AggressiveAI	28.72%	24.8%	6.60%	6.34%
ReleasedAI	70.64%	72.99%	33.57%	20.51%
DefaultAI	79.99%	77.22%	58.60%	22.39%
FlexibleAI	81.96%	84.04%	64.98%	60.42%

Figure 15: Victory rates of baseline AIs, plus the final FlexibleAI

However, most of this success came from a few simple sources. The biggest gains came just from the fact that the FlexibleAI made concentrated attacks with several suns and that the AI had acceptable parameters for danger and attack properties. Most other parameters were either surprisingly unimportant, such as the upgrade safety setting, or performed best in their simplest form, as evidenced by the fact that the AI found more success when soldiers ignored their target and instead moved directly towards the nearest enemy. Much of the potential cleverness of the FlexibleAI never proved to be useful.

More importantly, the nature of the testing framework itself presents a major problem with the resultant AI. Due to the setup of the tests and their scoring system, the AI is configured to best defeat this particular set of opponent AIs on this particular set of levels. This significantly biases the AI's optimal behavior. For example, one level starts with all three players already adjacent to each other. The AggressiveAI will never expand or upgrade in this scenario, and so the FlexibleAI tends to achieve either a very low victory rate or a perfect victory rate, depending on its own level of aggression. In another level, the players are situated on a circle and can only attack along that line, which limits tactical options. In this scenario, the resultant AI will perform worse than the DefaultAI. There is clearly room for situation-specific improvement, which the testing framework ignores, and the final AI is essentially tuned to best address only these opponents and levels with equal weight, which may not be representative of most live games.

The only true way to avoid this issue is to test the AI against its intended audience, i.e. human players. The logistical difficulties of getting enough data points to allow for conclusive analysis and iteration, however, were too great for this project.

8 Future Development

The FlexibleAI framework offers several possibilities for improvement. One such improvement would be a procedural analysis of the level in order to determine optimal parameters. As noted before, different configurations of the FlexibleAI would perform best on different levels, and the final AI parameters were merely the best average among them. If the AI were able to examine the level and determine a custom strategy, it may be able to improve its performance in all cases. Further, the FlexibleAI could dynamically alter its own parameters in the middle of a game, given sufficient guidelines.

The FlexibleAI also has the limitation of never considering the option of retreating. In some situation, a sun is surrounded and outnumbered, and the FlexibleAI will try to allocate it a full defense when the more advantageous move would be to forsake the sun and return its soldiers to another sun. Retreating can also be used offensively: by suddenly removing a large amount of soldiers from a valuable sun, a player can encourage the other two players to simultaneously attack the sun, annihilating each other while salvaging the majority of the first player's forces. The capability to recognize avoid such honey trap strategies could be equally valuable.

Another possible improvement would be some means of analyzing the flow of soldiers through a level. Since the entities in Auralux can be conceived of as sources, nodes, and sinks (enemies), it can be instructive to form strategies based on a perceived flow of soldiers and to consider what may match, disturb, or redirect that flow. This could require a significant departure from the FlexibleAI framework, however.

Other improvements could be added in the implementation of the real-time equivalent to the FlexibleAI, which I currently plan to build for the Auralux sequel. These would form the micromanagement component of the real-time AI and could allow it to quickly cancel attacks in the face of certain defeat, spatially concentrate attacks, or slow down attacks with harrying tactics.

My current plans for further development start with the transition of the FlexibleAI framework (informed by the tests performed in this project) to Auralux. It may be necessary to cause the FlexibleAI to act more aggressively or more variably for game design purposes, but I fully expect the core algorithm to offer a more challenging, less exploitable, and ultimately more fun AI opponent.